Ver. 1.3.0, 8/18/2023

Python Program for Product Operator Formalism of Spin-1/2 for Nuclear Magnetic Resonance

Author: Dr. Kosuke Ohgo

URL: https://github.com/ohgo1977/ProductOperator_Python

Email: please add @gmail.com to my GitHub account name

Purpose

This program is designed to handle the product operator formalism of spin-1/2 nuclei for Nuclear Magnetic Resonance (NMR) using Python, a free programming language widely used in scientific communities. The program can manipulate various types of operators, and this ability provides the user with a rich environment to perform calculations. The program will be helpful for educational use, for example, showing how to calculate product operators and explaining how pulse sequence components, such as Hahn-echo, INEPT etc., work. Also, the program can be used to calculate evolutions of a density operator under a pulse sequence including phase cycling that can be useful for research projects.

Requirements

This program was tested under Python 3.8.5 and 3.10.11 with SymPy (1.11.2) and NumPy (1.23.3). Basic knowledge of Python programming will be required.

Limitation

This program can only handle weakly-coupled spin-1/2 systems.

Note

This program is ported from the MATLAB version with Symbolic Math Toolbox (SMT). (https://github.com/ohgo1977/ProductOperator). The great benefit using Python is its availability in multiple platforms as a free programming language in contrast with commercial programming languages (e.g., MATLAB or Mathematica). As a disadvantage, the Python version is slower than the MATLAB version most likely due to a weaker capability of SymPy for symbolic calculations than the SMT. Also, the simplification of symbolic expressions with SymPy is trickier than the SMT. The details of the simplification are described in the Technical Details.

Introduction of Program

Before describing details, I will show a couple of examples demonstrating the ability and flexibility of this program.

The first example is how to create spin operators in the workspace. The command

```
>>> PO.create(['I', 'S'])
```

creates the variables, Ix, Iy, Iz, Ip, Im, Ia, Ib, Sx, Sy, Sz, Sp, Sm, Sa, Sb, and hE. They can be used as the spin angular momentum operators (I_x, I_y, I_z, …), lowering/raising operators (I^+, I^-, …), polarization operators (I^α, I^β, …) and a half of unity operator ($1/2E$) of the $I - S$ spin system. They are called **PO objects**.

Almost any product operators can be calculated by using these **PO objects**. For example, to calculate $[I_x, I_y] = I_xI_y - I_yI_x$,

```
>>> rho = Ix*Iy - Iy*Ix
```

The result is stored in the new **PO object**, `rho`, in this case. A **PO object** stores several types of information to describe product operators. They are called **PO properties**. As you can see the property named **txt**, the result is

```
>>> print(rho.txt)
Iz*(I)
```

that corresponds to the well-known equation $[I_x, I_y] = i*I_z$.

Another example is to get a matrix representation of $2I_xS_x - 2I_yS_y$,

```
>>> rho = 2*Ix*Sx - 2*Iy*Sy
```

Then, the matrix representation can be accessed via the property **M** by

```
>>> rhoMatrix = rho.M
[  0 0  0   1.0]
[  0 0  0   0  ]
[  0 0  0   0  ]
[1.0 0  0   0  ]
```

As you can see, it is a pure double quantum state. It is possible to express `rho` using the shift operators I^+, I^-, S^+ and S^-,

```
>>> rho_pmz = xyz2pmz(rho)
>>> print(rho_pmz)
IpSp + ImSm
```

Note that this line is equivalent to `print(rho_pmz.txt)`.

If there is a given matrix, the program can create a corresponding **PO object**. For example,

```
>>> M_in = [[1, 0], [0, -1]]
>>> rho = PO.M2xyz(M_in, ['I'])
>> print(rho)
Iz*(2)
```

The program can be used to check how NMR interactions influence a spin state, using dedicated functions called **PO methods**. For example, to see how I_z evolves under a $90°_y$ pulse followed by the chemical shift interaction,

```
>>> rho = Iz.pulse(['I'], ['y'], [pi/2]).cs(['I'], [o1*t])
Pulse: I  90y
```

```
    Ix
CS: I o1*t
    Ix*(cos(o1*t)) + Iy*(sin(o1*t))
```

The program has a flexibility to change the number of spins and spin labels. For example, if you would like to create the I_1-I_2-I_3 spin system,

```
>>> PO.create(['I1', 'I2', 'I3'])
```

then I1x, I1y, …, I3a, I3b, and hE will be created.

As shown above, the program provides various methods to construct and manipulate product operators that will be helpful for understanding the ideas of NMR spectroscopy.

0. How to Use Program

0.0. Path Setting

There are several ways to set up a path for a module. Please check Internet resources for detail. Here I will show an example of setting a path.

1. From a Python shell,

```
>>> import site
>>> site.getsitepackages()
```

then you will get information of site-packages, for example, C:¥Python38¥Lib¥site-packages.

2. Create a text file with an extension '.pth' in the directory in the step 1, for example, 'MyPath.pth'.

3. Write a path of a folder including '**PO.py**' in the file above (e.g., 'MyPath.pth').

For example,

C:¥Python38¥Lib¥site-packages¥PO_Python

where 'PO_Python' is the folder including 'PO.py', in this case.

0.1. Python Shell for Interactive Calculation

After opening PO.py in IDLE, Run > Run Module, then a Python shell for this program is ready. Run the **PO.create()** command to generate spin operators, e.g., **PO.create(['I', 'S'])** for the I-S spin system. Then, you can type in commands for further calculations.

An alternative way is to import the module. In that case,

```
>>> from PO import *
>>> PO.create(['I', 'S'])
```

```
>>> from PO import *
```

Note that the third command (the second call of '**`from PO import *`**') is a *necessary evil* to access PO objects and other symbolic variables generated by **PO.create()**. As a technical note, this issue is most likely related to the use of built-in **exec()** function to generate dynamically named PO objects and symbolic variables in the module. Please let me know if you know the tips to solve this issue.

0.2. Script File for Executing Multiple Commands

You can use a script file to run multiple lines in one time. A script file should include at least three lines below at the beginning followed by actual command lines.

```
from PO import *
PO.create(['I', 'S'])
from PO import *
```

As explained in the previous section, it is necessary to put the second call of '**`from PO import *`**' to access the PO objects and some symbolic variables. After these lines, you can describe multiple commands.

Then, the script file can be run in the terminal. For example, if you like to run Exp010_OnePulse.py,

```
> python Exp010_OnePulse.py
```

0.3. Script File for Running Pulse Sequence

As described in the section 3, the program can run a pulse sequence by loading a text input file describing parameters and NMR interactions. A pulse sequence will be executed using a function called **PO.run_PS()**. As an example, if the input file is 'Exp060_SpinEcho_PS.txt', the script file (e.g., 'Exp060_SpinEcho.py') is simply described as,

```
from PO import *
rho_cell, rho_detect_cell, rho_total, rho_obs, a0_M, rho_M =
PO.run_PS('Exp060_SpinEcho_PS.txt')
```

Then,

```
> python Exp060_SpinEcho.py
```

Note that only one line of '`from PO import *`' is required at the beginning of the script file in contrast with other type of scripts shown in the sections 0.2.

1. Creating Initial State of System

An initial state of a system, i.e., a density operator at the beginning, can be created from the combination of spin operators generated by a **PO** method called **PO.create()**.

PO.create(spin_label_cell) creates **PO objects** (spin operators) with labels defined in a list **spin_label_cell**. For example, in the case of the I-S two spin system,

```
>>> PO.create(['I', 'S'])
```

creates the **PO objects** Ix, Iy, Iz, Ip, Im, Sx, Sy, Sz, Sp, Sm, and hE in the workspace. It is possible to create a desired density operator by combining these **PO objects** with the '*', '+', '-', '/', and '^' operators and coefficients. Simultaneously, frequently used coefficients are created as symbolic variables with SymPy (see the explanation of **PO.symcoef()** for details of the coefficients created). As an example, to create $\rho = I_x*\cos\theta + 2I_yS_z*\sin\theta$,

```
>>> rho = Ix*cos(q) + 2*Iy*Sz*sin(q)
```

There are a couple of rules to use the '*', '+', '-', '/' and '^' operators with **PO objects**. Firstly, these operators should be used between **PO objects** with the same number of spin types. Rules for each operator are shown below. Hereafter, **obj** indicates a **PO object** unless otherwise noted.

- '*' operator can be used to calculate:

  1. **obj1*obj2**. If the **basis** properties of these objects are different, the rules below are applied to determine the basis of the returned object.

  Old bases New basis
  ('xyz', 'pmz') ➔ 'pmz'
  ('xyz', 'pol') ➔ 'pol'
  ('pmz', 'pol') ➔ 'pol'

  where 'xyz' is for the Cartesian operator basis (I_x, I_y, I_z), 'pmz' is for the lowering/raising operator basis (I^+, I^-, I_z), and 'pol' is for the polarization operator basis (I^α, I^β, I^+, I^-). These rules mean that the 'xyz' basis is overwritten with the 'pmz' or 'pol' bases, and the 'pmz' basis is overwritten with the 'pol' basis. If it is necessary to get the result with a different basis, use a basis-conversion method such as **pmz2xyz()**, **pol2xyz()** or **pol2pmz()**. For example, to obtain the result of Ix*Ia with the 'xyz' basis instead of the 'pol' basis,

  ```
  >>> Ix*PO.pol2xyz(Ia)
  ```

  2. **a*obj** and **obj*a**. **a** can be a number or symbolic.

- **'+'** operator can be used to calculate:

  1. **obj1 + obj2**. If the **basis** properties of these objects are different, the same rules for the '*' operator are applied.

2. **obj** + **a** and **a + obj**. **a** can be a number or symbolic. This calculation means an addition of **a*E** to **obj**.

- **'-'** operator can be used to calculate:

  1. **obj1 - obj2**. If the **basis** properties of these objects are different, the same rules for the '*' operator are applied.

  2. **obj - a** and **a - obj**. **a** can be a number or symbolic. This calculation means **obj - a*E** and **a*E - obj**, respectively.

- **'/'** operator can be used to calculate:

  obj/a. **a** can be a number or symbolic. Note that a **PO object** cannot be a divisor, i.e., **a/obj** or **obj1/obj2** cannot be calculated.

- **'^'** operator can be used to calculate:

  obj^n. **n** should be 0 or a positive integer.

As mentioned above, information characterizing current product operators are stored as **PO properties**. Useful properties for users are '**txt**' that shows a text output of the operators, '**M**' that shows a matrix representation of the operators, '**coherence**' that shows populations and coherences in the matrix, and **logs** that keeps the record of the applied methods. Values of a **PO property** can be obtained by the syntax **obj.PropertyName**. For example, to get a matrix representation,

```
>>> rho_matrix = rho.M
```

2. Applying NMR Interactions to System

Effects of NMR interactions such as RF pulse, chemical shift and *J*-coupling to a spin system can be calculated by **PO methods**.

2.1. RF Pulses

The method to apply a single pulse or simultaneous pulses is

obj = PO.pulse(obj, sp_cell, ph_cell, q_cell) or

obj = obj.pulse(sp_cell, ph_cell, q_cell) ,

where **sp_cell**, **ph_cell**, and **q_cell** are lists describing spin types to be manipulated, quadrature phases for pulses, and flip angles of pulses in radian, respectively. An element of **sp_cell** can be characters ('I', 'S', 'I1' or 'I2' etc. defined in **spin_label**) or the numbers describing the order of the spins in **spin_label** (1 for 'I', 2 for 'S' in ['I', 'S'] etc.). An element of **ph_cell** can be the characters such as 'x', 'X' or '-y' or the numbers 0, 1, 2 or 3 for x, y, -x or -y, respectively. An element of **q_cell** can be a number or symbolic.

Examples

```
>>> PO.create(['I', 'S'])
>>> rho = PO.pulse(Iz, ['I'], ['x'], [pi/2])
```

or equivalently,

```
>>> rho = Iz.pulse([1], [0], [pi/2])
```

```
>>> rho = Iz + Sz
>>> rho = PO.pulse(rho, ['I', 'S'], ['x', 'y'], [pi/2, pi/2])
```

It applies a 90_x pulse to I and 90_y to S. Equivalently,

```
>> rho = PO.pulse(rho, [1, 2], [0, 1], [pi/2, pi/2])
```

The wildcard character '*****' can be used for **sp_cell** to make an input line simple. Let's assume a 5-spin system, I_1, I_2, I_3, S_4 and S_5.

```
>>> PO.create(['I1', 'I2', 'I3', 'S4', 'S5'])
>>> rho = I1z + I2z + I3z + S4z + S5z
```

If applying a 90_x pulse to all *I* spins, the wildcard character can be used as

```
>>> rho = PO.pulse(rho, ['I*'], ['x'],[pi/2])
```

If applying 90_x pulses to both *I* and *S* spins,

```
>>> rho = PO.pulse(rho, ['*'], ['x'], [pi/2])
```

If applying a 90_x pulse to all *I* spins and a 180_y pulse to all *S* spins,

```
>>> rho = PO.pulse(rho, ['I*', 'S*'], ['x', 'y'], [pi/2, pi])
```

Note that use of '*' shown below may cause a confusion.

```
>>> PO.create(['I','S'])
>>> rho = PO.pulse(Iz + Sz, ['*'], ['x','y'], [pi/2, pi])
```

returns

```
Pulse: I  90x
   Sz + Iy*(-1)
Pulse: S  90x
   Sy*(-1) + Iy*(-1)
```

In this case, the phase ‘x’ and flip angle pi/2 were applied on both I and S spins.

Pulses with Phase Shift

The method to apply a single pulse or simultaneous pulses with arbitrary phases is

obj = PO.pulse_phshift(obj, sp_cell, ph_cell, q_cell) or

obj = obj. pulse_phshift(sp_cell, ph_cell, q_cell) .

The difference from **pulse()** is that **ph_cell** includes arbitrary phases in radian. An element of **ph_cell** can be a number or symbolic.

2.2. Chemical Shift

The method to apply the chemical shift evolution to spins is

obj = PO.cs(obj, sp_cell, q_cell) or

obj = obj. cs(sp_cell, q_cell)

where **sp_cell** and **q_cell** are cell arrays describing spin types to be affected and rotation angles in radian, respectively. The formats of **sp_cell** and **q_cell** are same as the ones used for **pulse()**. The wildcard character **'*'** also can be used for **sp_cell** same as **pulse()**.

```
>>> PO.create(['I1', 'I2', 'S3'])
>>> rho = I1x + I2x + S3x
>>> rho = PO.cs(rho, ['I1'], [oI*t])
>>> rho = PO.cs(rho, ['I2'], [oI*t])
>>> rho = PO.cs(rho, ['S3'], [oS*t])
```

or equivalently,

```
>>> rho = PO.cs(rho, [1, 2, 3], [oI*t, oI*t, oS*t])
```

or

```
>>> rho = PO.cs(rho,['I*', 'S3'], [oI*t, oS*t])
```

2.3. *J*-coupling

The method to apply the *J*-coupling evolutions to spin pairs is

obj = PO.jc(obj, sp_cell, q_cell) or

obj = obj.jc(sp_cell, q_cell) ,

where **sp_cell** and **q_cell** are cell arrays describing labels for the spin pairs corresponding to the *J*-coupling Hamiltonians and rotation angles in radian, respectively. An element of **sp_cell** can be characters 'IS', 'I1I3' etc. or a 1 x 2 list showing the index of spins such as [1, 2] or [1, 3].

```
>>> PO.create(['I', 'S'])
>>> rho = Ix
>>> rho = PO.jc(rho, ['IS'], [pi*J12*t])
```

Note that the wildcard character **'*'** CANNOT be used for **sp_cell** in this method. The spin pairs must be explicitly given in **sp_cell**.

```
>>> PO.create(['I1', 'I2', 'S3'])
>>> rho = I1x + I2x
>>> rho = PO.jc(rho,['I1S3', 'I2S3'], [pi*J13*t, pi*J23*t])
```

2.4. Pulse Field Gradient

The method to apply a pulse field gradient is

obj = PO.pfg(obj, G_coef, gamma_cell) or

obj = obj.pfg(G_coef, gamma_cell) ,

where **G_coef** is a ratio of the gradient field strength and **gamma_cell** is a cell array to store gyromagnetic ratios of spins in the system.

G_coef can be a number or symbolic. Components of **gamma_cell** can be also a number or symbolic.

```
>>> PO.create(['I1', 'I2', 'S3'])
>>> rho = I1x + I2x + S3x
>>> rho = PO.pfg(rho, 1, [gH, gH, gC])
```

Internally, angles are calculated from **G_coef**, **gamma_cell**, and the internal, symbolic constant **GZ** as their products, and they are used as input parameters of **cs()**. In the case above, the angles are `[GZ*gH, GZ*gH, GZ*gC]`. Note that a length of the gradient pulse is not considered in the calculation. If necessary, involve a time constant into **G_coef** (e.g., `t`). This method is obtained from the reference (Güntert, 2006).

The method to delete terms influenced by a pulse filed gradient is

obj = PO.dephase(obj) or

obj = obj.dephase ().

This method is obtained from the reference (Güntert, 2006).

```
>>> PO.create(['I1', 'I2', 'S3'])
>>> rho = I1x + I2x + S3z
>>> rho = PO.pfg(rho, 1, [gH, gH, gC])
>>> rho = PO.dephase(rho)
```

2.5. Notes for Applying PO Methods to PO Object

Independence of Methods from Basis Type

The methods shown above can be applied to a **PO object** with any basis type. The basis type of the input **PO object** is applied to the resulting **PO object**. For example,

```
>>> PO.create(['I', 'S'])
>>> PO.pulse(PO.xyz2pmz(Iz), ['I'], ['y'], [pi/2]) # pmz basis
Pulse: I  90y
    Ip*(1/2) + Im*(1/2)
>>> PO.pulse(PO.xyz2pol(Iz), ['I'], ['y'], [pi/2]) # pol basis
Pulse: I  90y
    IpSa*(1/2) + IpSb*(1/2) + ImSa*(1/2) + ImSb*(1/2)
```

Applying Multiple PO Methods in One Line

You can apply multiple methods in one line using the dot '**.**' as a separator between methods,

```
>>> PO.create(['I', 'S'])
>>> rho = Iz.pulse(['I'],['y'],[pi/2]).cs(['I'],[q]).jc(['IS'],[pi*J12*t])
```

that is equivalent to

```
>>> rho = Iz.pulse(['I'], ['y'], [pi/2])
>>> rho = rho.cs(['I'], [q])
>>> rho = rho.jc(['IS'], [pi*J12*t])
```

Note that the order of the methods to be applied to the system is left to right (**pulse()** => **cs()** => **jc()**) but not right to left (**jc()** => **cs()** => **pulse()**).

3. Running Pulse Sequence

It is possible to construct a pulse sequence combining the **PO methods** above in addition to other **PO methods** as utilities (**4. Utilities** for details). In this program, a format for constructing a pulse sequence is provided with the dedicated **PO method**, **PO.run_PS()**. Many types of pulse sequences can be calculated using them.

PO.run_PS(fname)

runs a pulse sequence defined in a text file, **fname**. Note that it is not necessary for the text file to be a Python script file (*.py). Any ASCII file can be loaded. The lines below show an example of the input file (refocused INEPT with two-step phase cycling).

```
# Para begin #
spin_label_cell = ['I', 'S']
rho_ini = Iz*B + Sz
obs_cell = ['S']
no_ph = 7 # The number of pulse phases used in the pulse sequence
ph_cell[1] = [0,2] # ph1
ph_cell[2] = [0]   # ph2
ph_cell[3] = [0]   # ph3
ph_cell[4] = [0]   # ph4
ph_cell[5] = [1]   # ph5
ph_cell[6] = [0]   # ph6
ph_cell[7] = [0]   # ph7
phRtab = [0,2]
phid = list(range(2)) # phid: 0-based index
coef_cell = ['rINET_flag']
```

```
disp_bin = 1
# Para end #

# PS begin #
print(type(rINET_flag))
rho = rho.pulse(['I'], [ph1], [1/2*pi])
rho = rho.pulse(['I', 'S'], [ph3, ph2], [pi, pi])
rho = rho.jc(['IS'], [pi*J*2*t1])
rho = rho.pulse(['I', 'S'],[ph5, ph4],[1/2*pi, 1/2*pi])
rho = rho.pulse(['I', 'S'],[ph7, ph6],[pi, pi])
rho = rho.jc(['IS'], [pi*J*2*t2])
# PS end #
```

The parameters required for the pulse sequence should be described between the lines, '`# Para begin #`' and '`# Para end #`'.

spin_label_cell is a cell array for the spin labels. If it is not assigned, ['I', 'S', 'K', 'L', 'M'] is used, meaning that a 5-spin system is created, and the calculation speed will get slow. Try to assign **spin_label_cell** explicitly.

rho_ini is a **PO object** of the initial state. It should be described by **PO objects**.

obs_cell is a cell array defining the observed spins used in **observable()**. The wildcard character '*' can be used. If it is not assigned, ['*'] is used, meaning all spins are observed.

no_ph is the number of pulse phases used in the pulse sequence.

ph_cell is a list defining the phase tables. ph_cell[*n*] is a list corresponding to ph*n* in the pulse sequence. Describe ph_cell[1], ph_cell[2], …, for ph1, ph2, …, respectively. Note that ph0 is not created from ph_cell[0].
phRtab is a list defining the receiver phase. Elements can be a quadrature or arbitrary phase.

phid is a list defining the phase cycling steps expressed in the 0-based index. It is not necessary to assign **phid** if a full phase cycling is used. **phid** should be explicitly assigned if it is necessary to run particular sets of phase steps. For example, if you like to check 2nd and 4th steps of the phase cycling, phid = [1, 3] with the 0-based index.

coef_cell is a list of string elements describing additional symbolic coefficients not created by **PO.symcoef()**. If it is not assigned, [] is used.

disp_bin is a value to control the display of the pulse sequence on the Command Window (1 for ON, 0 for OFF). If it is not assigned, 1 is used.

The code is not sensitive to the order that the above parameters are described. If there are additional parameters required for the pulse sequence section, they should be described in this section. Note that if a parameter is created by another parameter, e.g., a1 = 2 and a2 = 2*a1, they should be described in that order.

The section for the pulse sequence should be described between the lines, '`# PS begin #`' and '`# PS end #`'. Names of the input and output **PO objects** should be **rho**. The code is sensitive to the order of the methods described in this section.

The output parameters, **rho_cell, rho_detect_cell, rho_total, rho_obs, a0_M, and rho_M** are explained below.

rho_cell is a list storing **PO objects** after the pulse sequence section for phase cycling steps. For example, if there are 8 steps of the phase cycling, **rho_cell** stores 8 **PO objects**.
In each phase cycle, the resulting **PO object** after the pulse sequence section is handled by **receiver()**. The obtained **PO object** from **receiver()** is stored in **rho_detect_cell**.

rho_total is a **PO object** that is the sum of **rho_detect_cell**.

rho_obs is a **PO object** that shows observable spins in **rho_total**, obtained from **rho_obs = observable(rho_total, obs_cell)**.

a0_M and **rho_M** are matrices combining **a0_V** and **rho_V** from **[a0_V, rho_V] = SigAmp(obj, obs_cell, phR),** where **obj** is a **PO object** after the pulse sequence section.

4. Utilities

There are additional **PO methods** as utilities. Some of them are **static methods**, i.e., they are called by the syntax **PO.MethodName()**. These utility methods can be used, for example, when a pulse sequence is simulated.

obj_pmz = PO.xyz2pmz(obj_xyz) or
obj_pmz = obj_xyz.xyz2pmz()
and
obj_xyz = PO.pmz2xyz(obj_pmz) or
obj_xyz = obj_pmz.pmz2xyz()
are for the conversions between the Cartesian operator basis (**obj_xyz)** and raising/lowering operator basis (**obj_pmz**) (Güntert, 2006). **xyz2pol()** and **pol2xyz()** are for the conversion between the 'xyz' and 'pol' bases, and **pmz2pol()** and **pol2pmz()** are for the conversion between the 'pmz' and 'pol' bases.

obj = PO.set_basis(obj, basis_in) or
obj = obj.set_basis (basis_in)
rewrites **obj** to a different basis **basis_in**.

obj_pol = PO.M2pol(M_in, spin_label_cell)
obj_pol, a **PO object** with the 'pol' basis, is created from a matrix **M_in**. **M_in** should be 2^N x 2^N in the double or sym class. **spin_label_cell** is a cell array for spin labels. If **spin_label_cell** is not assigned, ['I', 'S', 'K', 'L', 'M'] is used for the spin label. Similarly, **obj_pmz = PO.M2pmz(M_in, spin_label_cell)** and **obj_xyz = PO.M2xyz(M_in, spin_label_cell)** are used to create **PO objects** with the 'pmz' and 'xyz' bases, respectively, from **M_in**.

PO.symcoef(spin_label_cell, add_cell)
creates frequently used symbolic constants based on the information of **spin_label_cell**. For example,
```
>>> PO.symcoef(['I', 'S'])
```
creates oI, oS, o1, o2, JIS, J12, gI, gS, g1, g2 systematically from the list `['I', 'S']` in addition to a, b, c, d, f, gH, gC, gN, t1, t2, t3, t4, t, q, w, B, J, and G. It is possible to add other parameters by **add_cell** as a list with string, e.g., `['a1', 'b1']` for adding symbolic variables a1 and b1.

PO.dispPO(obj) or
obj.dispPO()
displays the terms in **obj** in the following manner.
ID Product-Operator Coefficient
For example, in the case of $\rho = I_x*\cos\theta + I_y* \sin\theta$

```
>>> PO.create(['I'])
>>> rho = Ix*cos(q) + Iy*sin(q)
>>> rho.dispPO()
ID  Term                  Coef
1       Ix               cos(q)
2       Iy               sin(q)
```

Note that the index shown is the 1-based index.

obj = PO.receiver(obj, phR) or

obj = obj. receiver(phR)

rotates product operators (defined by **obj) -phR** around Z-axis where **phR** is a receiver phase in the quadrature (i.e., 'x', 'X', 0, '-y', '-Y', 3, etc.) or arbitrary phase. This method is obtained from the reference (Güntert, 2006).

obj = PO.observable(obj, sp_cell) or

obj = obj. observable (sp_cell)

selects observable terms of spin types defined by a list **sp_cell**. The format of **sp_cell** is same as the one used for **pulse()** accepting the wildcard character '*', e.g., ['I*']. If **sp_cell** is not assigned, all spin types are considered for the selection. This method is obtained from the reference (Güntert, 2006).

[a0_V,rho_V] = PO.SigAmp(obj, sp_cell, phR) or

[a0_V,rho_V] = obj.SigAmp(sp_cell, phR)

calculates a signal amplitude $a_{[-]}$ corresponding to a (-1) quantum coherence in the equation

$a_{[-]} = 2*i*\rho_{[-]}(0) *\exp(-i*\phi_{rec})$

in *Spin Dynamics* (2nd Ed.), p. 288 (eq. 11.48).

For example, in the case of a homonuclear 2-spin system, the equation is

$[a_{[-\beta]} \quad a_{[-\alpha]} \quad a_{[\beta-]} \quad a_{[\alpha-]}] = 2*i*[\rho_{[-\beta]}(0) \quad \rho_{[-\alpha]}(0) \quad \rho_{[\beta-]}(0) \quad \rho_{[\alpha-]}(0)]*\exp(-i*\phi_{rec})$

as shown in p. 379 of the reference.

Related topics: *Spin Dynamics* (2nd Ed.), p.262, p. 287, p. 371, p.379, pp.608-610.

sp_cell is a list describing the spin types to be observed (e.g., ['I'], ['I', 'S'], ['I1', 'I2'], [1], [1, 2]). The wildcard character **'*'** can be used for **sp_cell** same as **pulse()**.

phR is a quadrature receiver phase (e.g. 'x', 'y', 0, 1) or arbitrary value (expect for 0, 1, 2, 3 as they are considered as a quadrature phase).

a0_V is a list corresponding to $2*i*[\rho_{[-\beta\ldots]}(0)\ \rho_{[-\alpha\ldots]}(0)\ \rho_{[\beta-\ldots]}(0)\ \rho_{[\alpha-\ldots]}(0)\ \ldots] *\exp(-i*\phi_{rec})$.

rho_V is a list describing which component of **a0_V** corresponds to which coherence in the density operator (e.g., 'ua' for '+α', 'bd' for 'β-').

phout = PO.phmod(phx, ii)

outputs a phase value, **phout**, from a phase table (list), **phx**. If **ii** is smaller to the length of **phx**, the phout = phx(ii) otherwise phout = phx(mod(ii, length(phx)). This method can be used in cases where there are phase tables with different lengths and phase-cycle them together.

Example

ph1 = [0, 1, 2, 3] # 4 steps

ph2 = [0, 1, 2, 3, 2, 3, 0, 1] # 8 steps

To phase-cycle them together, 8 steps are necessary. In such case, **PO.phmod(ph1, ii)** returns

ii = 0 ➔ 0, ii = 1 ➔ 1, ii = 2 ➔ 2, ii = 3 ➔ 3, ii = 4 ➔ 0, ii = 5 ➔ 1, ii = 6 ➔ 2, ii = 7 ➔ 3

while **PO.phmod(ph2, ii)** returns

ii = 0 ➔ 0, ii = 1 ➔ 1, ii = 2 ➔ 2, ii = 3 ➔ 3, ii = 4 ➔ 2, ii = 5 ➔ 3, ii = 6 ➔ 2, ii = 7 ➔ 1 .

obj = PO.delPO(obj, id_in) or

obj = obj.delPO(id_in)

deletes terms from **obj** using the information given by **id_in**.

If **id_in** is a list including numbers, these numbers are indexes for terms to be deleted. The index number of each term can be found by **dispPO()**. Note that this index is the 1-based index, not 0-based index that is used in Python as a default. The example is

```
>>> PO.create(['I', 'S', 'K'])
>>> rho = Ix + Sx + Kx
>>> rho.dispPO()
ID  Term                 Coef
1       Kx               1
2       Sx               1
3       Ix               1
>>> rho = rho.delPO([1, 2]) # Deleting Kx and Sx
>>> rho.dispPO()
ID  Term                 Coef
1       Ix               1
```

If **id_in** is a list with strings describing product operators, terms for these operators are deleted.

There are some examples.

```
>>> rho = PO.delPO(rho,['Ix']) # delete Ix term
>>> rho = PO.delPO(rho,['I2y']) # delete I2y term
>>> rho = PO.delPO(rho,['IzSz']) # delete 2IzSz term
>>> rho = PO.delPO(rho,['Ix', 'IzSz'])# delete Ix and 2IzSz term
```

The wildcard character '*****' can be used to choose all three phases.

```
>>> rho = PO.delPO(rho,['IxS*'])# delete 2IxSx, 2IxSy and 2IxSz if they exist
>>> rho = PO.delPO(rho,['I*S*', 'I*S*K*'])# delete any terms including 2I*S*
and 4I*S*K*
```

In a spin system with spin names with numbers, i.e., I1, I2, S3 etc, the wild character should be used after the spin names. For example

```
>>> rho = PO.delPO(rho,['I1*', 'I1*S3*'])# delete terms including I1* and 2I1*S3*.
Note that terms with I1*I2*S3* will remain.
```

obj = PO.selPO(obj, id_in) or

obj = obj.selPO(id_in)

selects terms from **obj** using the information given by **id_in**. The format of **id_in** is same as in **delPO()**.

id_vec = PO.findcoef (obj, coef_in_cell) or

id_vec = obj. findcoef (coef_in_cell)

finds particular terms that includes the coefficients defined in a list **coef_in_cell** and returns index numbers for these terms as a list **id_vec**. The index of **id_vec** is the 1-based index. This function can be used with **delPO()** or **selPO()** to delete or select certain terms.

For example,

```
>>> PO.create(['I'])
>>> rho = Ix*cos(q) + Iy*sin(q)
>>> rho.dispPO()
ID  Term                  Coef
1       Ix                cos(q)
2       Iy                sin(q)

>>> id_vec = rho.findcoef([cos(q)])
```

returns [1].

obj = PO. simplify_cos (obj) or

obj = obj. simplify_cos()

or

obj = PO. simplify_exp (obj) or

obj = obj. simplify_exp()

simplifies the coefficients of obj using rewrite(cos).simplify() or rewrite(exp).simplify(). This function can be used if the calculated coefficients are complexed.

Technical Details

Design of Program

The code is written with the object-oriented programming (OOP) style. In manner of OOP, parameters for characterizing product operators and functions for NMR interactions are called **properties** and **methods** of a **class** named **PO**, respectively. A **PO class object** stores the **PO class properties**, and the **object** is processed by the **PO class methods**. By designing **properties** and **methods** properly, **PO class objects** can be handled in manner of the product operator formalism. For example, the '*', '+', '-' , '/' and '^' operators are implemented as **PO methods** so that they work as corresponding operators for product operators (it is called operator overloading).

PO Class Properties

Any product operator can be described by three characteristic properties, spin types, axis labels (x, y or z) and a coefficient. For example, in the case of $-4I_xS_zK_z*\cos\theta$, the axis labels are x, z and z for the 1st (I), 2nd (S) and 3rd (K) spins, respectively, and the coefficient is $-\cos\theta$. Note that the coefficient "4" is related to the number of the active spin types in the product operator (in this case the number (Ns) is 3 for I, S and K, and 4 can be obtained from 2^{Ns-1}) and thus it is not considered as an independent coefficient. In the **PO class properties**, information on the axis labels for the spins and the coefficients are stored as **axis** and **coef**, respectively.

axis: This property stores axis labels of product operators as a NumPy matrix. It is a M x N matrix where M is the number of product operators in the system and N is the number of spin types in the system. Column positions of the matrix correspond to the spin types. Each component in the matrix has a value of 0, 1, 2 or 3 corresponding to E, I_x, I_y or I_z, respectively, in the case of the Cartesian basis. In the case of the lowering/raising operator basis, 4 and 5 are used for I^+ and I^-, respectively. Also, in the case of the polarization operator basis, 6 and 7 are used for I^α and I^β, respectively. For example, the **axis** property of $I_{1x} + I_{2x} + I_{2z}$ (M = 3) in the I_1-I_2 system (N = 2) is matrix([[0, 1], [0, 3], [1, 0]])

coef: This property stores coefficients including signs for product operators as a SymPy Matrix. It is a M x 1 vector. Note that the 2^{Ns-1} coefficient at the beginning of a product operator is stored in another property, **Ncoef**.

There are additional properties in the **PO class**.

txt: This property stores a text output of the current system. It is automatically generated. As a default, the asterisk '*' is not displayed between spin operators and between the 2^{Ns-1} coefficient and a spin operator, e.g., 2IxSy*a. By changing a class variable called **asterisk_bin** to 1, '*' is displayed, e.g., 2*Ix*Sy*a. This variable can be changed by rewriting the code. The change of this property affects all **PO objects**. The former makes the **txt** and **logs** properties good looks while the later can be used to re-create a **PO object** from the text information stored in the **logs** property. This property can be displayed by **obj.txt** or **print(obj)**.

spin_label: This property defines the labels of the spin types in a system such as, I,S,K, … or I1, I2, I3, … etc.

basis: This property stores a type of the operator basis of the current system. There are three types of bases, ‘xyz’ for the Cartesian operator basis (Ix, Iy, Iz), ‘pmz’ for the lowering/raising operator basis (Ip, Im, Iz) and ‘pol’ for the polarization operator basis (Ia, Ib, Ip, Im).

disp: This property stores values of 1 or 0 to control the output display of the applied method and the calculated result on the command window. The default value is 1 for display 'ON'.

logs: This property stores logs of methods applied to the **PO object**. Note that operations with the ‘*’, ‘+’, ‘-‘ , ‘/’ and ‘^’ operators overwrite the previous logs to the current **obj.txt**.

Ncoef: This property stores the $2^{N_S - 1}$ coefficients for product operators in the current system as a NumPy array. It is automatically calculated from the **axis** property.

M: This property stores a matrix representation of the current system as a SymPy Matrix. It is automatically generated.

coherence: This property is a 2^N x 2^N matrix displaying populations of spin states on the diagonal and coherences between states on the off-diagonal. *a* and *b* mean |α> and |β> states, respectively, and *d* and *u* mean coherences of |α> => |β> and |β> => |α>, respectively. Note that *d* and *u* are used in this display instead of *m* and *p*. The reason is that a symbol including *bm* is interpreted as a bold format in printing. For example, *abm* becomes **a**.

Merits to Use Axis Property

The **axis** property is beneficial for some important calculations in this program. One is a multiplication of **PO objects**. This calculation can be handled as an addition of the **axis** properties of the **PO objects**. For example, **axis** properties of Iz, Sy, Kx and their product Iz*Sy*Kx are:

```
Iz      : [3, 0, 0]
Sy      : [0, 2, 0]
Kx      : [0, 0, 1]
Iz*Sy*Kz: [3, 2, 1]
```

As you can see, the **axis** property of Iz*Sy*Kx is the sum of the three vectors (i.e., the **axis** property) of the three operators.

As a note, there is an exception for a multiplication of the same spin type. For example, if you consider the product of Iz, Ix*Sy and Kx,

```
Iz          : [3, 0, 0]
IxSy        : [1, 2, 0]
Kx          : [0, 0, 1]
```

`Iz*Ix*Sy*Kz: [4, 2, 1]# Sum of three vectors. It should be [2, 2, 1]!`

In this case, a special calculation is necessary for the *I*-spin because the **axis** value obtained by the addition is not correct. The **axial** value of Iz*Ix should be 2 instead of 4 because $I_z*I_x = i/2*I_y$. In the code, there is a branch to calculate an appropriate **axis** value and a coefficient for this type of cases.

Another benefit is that **axis** values of two operators can be used as indexes of a matrix that describes the cyclic commutations of the two operators. The details are explained in the next section.

Cyclic Commutation Rules

If operators *A*, *B*, and *C* shows $[A, B] = iC$, $[B, C] = iA$ and $[C, A] = iB$ (cyclic commutation) then

$\exp(-i\theta A)\, B \exp(i\theta A) = B\cos\theta + C\sin\theta$,

$\exp(-i\theta B)\, C \exp(i\theta B) = C\cos\theta + A\sin\theta$, and

$\exp(-i\theta C)\, A \exp(i\theta C) = A\cos\theta + B\sin\theta$

It is known that the spin angular momentum operators I_x, I_y, I_z are in cyclic commutation ($[I_z, I_x] = iI_y$) and the formula above can be used to describe an evolution of a density operator under a Hamiltonian. For example, a density operator $\rho(0) = I_x$ evolves under a chemical shift Hamiltonian $H = \omega I_z$ during a time period of t as

$\rho(t) = \exp(-iHt)\, \rho(0) \exp(iHt) = \exp(-i\omega t I_z)\, I_x \exp(i\omega t I_z) = I_x\cos(\omega t) + I_y\sin(\omega t)$

Use of Master Table to Accelerate Calculation

			B		
			x	y	z
			1	2	3
A	x	1	**0**	**3**	**-2**
	y	2	**-3**	**0**	**1**
	z	3	**2**	**1**	**0**
			C		

The cyclic commutation rules can be summarized as a table (master table) using the equation $\exp(-i\theta A)\, B \exp(i\theta A) = B\cos\theta + C\sin\theta$ above and I_x, I_y and I_z. For example, in the case of $\exp(-i\theta I_z)\, I_x \exp(i\theta I_z) = I_x\cos\theta + I_y\sin\theta$, the axis numbers of the *A* and *B* positions are 3 (z) and 1 (x), respectively. Then the axis number for *C* is the 3rd-row, 1st-column component in the table, i.e., 2 meaning I_y. If the value for *C* is 0 for given *A* and *B*, then *A* and *B* are not in the cyclic commutation (e.g., $A = B = I_x$). If the value for *C* is negative, then $-C\sin\theta$ is used instead of $+C\sin\theta$. This is the basic idea of the calculation in the code. The calculation using the master table is much faster than the matrix calculation (e.g., `exp(-I*q*Iz.M)*Ix.M*exp(I*q*Iz.M)`).

When can Master Table be Used for Calculation?

If the two rules below are satisfied, an evolution of ρ under *H* can be calculated by using the master table. Otherwise ρ does not evolve under *H*.

Rule 1. There should be at least one spin type matching between *H* and ρ.

AND

Rule 2. Only one spin type in the matching spin types has different axis labels between *H* and ρ.

Note that these rules can be used for spin-1/2 with the condition that one of H and ρ is a product of up to two spin operators (e.g. $H = 2I_zS_z$ but not like $4I_zS_zK_z$). Since the Hamiltonians of the pulse, chemical shift evolution, and J-coupling evolution satisfy this condition naturally, the master table can be used for the calculation. An example that doesn't satisfy Rule 1 and 2 but satisfies the cyclic commutation are the sets of $4I_xS_yK_z$, $4I_yS_zK_x$ and $4I_zS_xK_y$.

Rule 1 is obvious but how about Rule 2. Here is the analysis. Suppose $H = 2I_aS_b$ and $\rho = 8I_bS_bKL$ where K, L are spin operators that are different types each other in addition to I and S. Suppose $[I_a, I_b] = iI_c$ in the cyclic commutation. There are two spin-types matching between H and ρ, thus satisfying Rule 1, and only one of them (I spin) has different labels between H and ρ, thus satisfying Rule 2.

Then $[H, \rho] = 2I_aS_b\ 8I_bS_bKL - 8I_bS_bKL\ 2I_aS_b = 16\ I_aI_bS_b^2KL - 16I_bI_aS_b^2KL$
$= 16\ (I_aI_b - I_bI_a)S_b^2KL = 4(I_aI_b - I_bI_a)KL = i4I_cKL$. Note that $S_b^2 = 1/4E$ for spin-1/2.

$[4I_cKL, H] = 4I_cKL\ 2I_aS_b - 2I_aS_b\ 4I_cKL = 8I_cI_aS_bKL - 8I_aI_cS_bKL = 8(I_cI_a - I_aI_c)S_bKL = i8I_bS_bKL = i\rho$.

$[\rho, 4I_cKL] = 8I_bS_bKL\ 4I_cKL - 4I_cKL\ 8I_bS_bKL = 32I_bI_cS_bK^2L^2 - 32I_cI_bS_b\ K^2L^2 = 2(I_bI_c - I_cI_b)\ S_b = i2I_aS_b = iH$. Note that $K^2 = L^2 = 1/4E$ for spin-1/2.
Thus, H and ρ are in the cyclic commutation.

What if Rule 2 is not satisfied? In the case of $H = 2I_bS_b$ and $\rho = 8I_bS_bKL$,
$[H, \rho] = 2I_bS_b\ 8I_bS_bKL - 8I_bS_bKL\ 2I_bS_b = 16\ I_b^2S_b^2KL - 16I_b^2S_b^2KL = 0$
thus, H and ρ are not in the cyclic commutation.

In the case of $H = 2I_aS_b$ and $\rho = 8I_bS_cKL$, $[H, \rho]$ is calculated as 0 from $[2I_aS_b, 2I_bS_c] = 0$ that can be obtained from the matrix representation.
$[H, \rho] = 2I_aS_b\ 8I_bS_cKL - 8I_bS_cKL\ 2I_aS_b = 2I_aS_b\ 2I_bS_c\ 4KL - 2I_bS_c\ 4KL\ 2I_aS_b = 2I_aS_b\ 2I_bS_c\ 4KL - 2I_bS_c\ 2I_aS_b\ 4KL = [2I_aS_b, 2I_bS_c]*4KL = 0$ (See *Spin Dynamics* (2nd Ed.), p. 403, Eq. 15.24 and p. 407, Note 3).
Examples for these rules
$\rho = I_y$ and $H = I_z$
Both ρ and H include I-spin (Rule 1: yes) and they have different axis labels (I_y vs. I_z) (Rule 2: yes). ➔ Master Table: Yes

$\rho = S_y$ and $H = I_z$
ρ and H don't have a same type of spin (Rule 1: No) ➔ Master Table: No

$\rho = I_y$ and $H = I_y$

Both ρ and *H* include *I*-spin (Rule 1: yes) but they have the same axis label (I_y) (Rule 2: no). ➔ Master Table: no

$\rho = 2I_zS_y$ and $H = I_z$

Both ρ and *H* include I-spin (Rule 1: yes) but *I*-spins have the same axis label (I_z) (Rule 2: no). ➔ Master Table: no

$\rho = 2I_zS_y$ and $H = 2I_zS_z$

Both ρ and *H* include *I*- and *S*-type product operators (Rule 1: yes) and only *S*-spins have different axis labels (S_y vs. S_z) (Rule 2: yes). ➔ Master Table: yes

$\rho = 2I_xS_x$ and $H = I_z$

Both ρ and *H* include *I*-spin (Rule 1: yes) and they have different axis labels (I_x vs. I_z) (Rule 2: yes). ➔ Master Table: Yes

$\rho = 2I_xS_x$ and $H = 2I_zS_z$

Both ρ and *H* include *I*- and *S*-type product operators (Rule 1: yes) but both spin types have different axis labels (I_x vs. I_z and S_x vs. S_z) (Rule 2: no). ➔ Master Table: No

According to *Spin Dynamics* (2nd Ed.), p. 483, there are four cases where a product operator does not evolve under a *J*-coupling Hamiltonian I_jzI_kz.

Case 1. If both spin I_j and I_k are missing in the product operator.

Case 2. If only one spin I_j or I_k is present, and that spin carries a z label.

Case 3. If both spins I_j and I_k are present, but both spins carry a z label.

Case 4. If both spins I_j and I_k are present, but neither spin carries a z label.

These all four cases are excluded by the two rules above.

Case 1. Rule 1 is not satisfied.

Case 2. Rule 1 is satisfied but Rule 2 is not satisfied.

Case 3. Rule 1 is satisfied but Rule 2 is not satisfied.

Case 4. Rule 1 is satisfied but Rule 2 is not satisfied.

Simplification of Coefficients with SymPy (ver. 1.3 ~)

The simplification of mathematical expressions is an important process of this program to show results of the calculations easily readable with keeping a physical meaning. However, what does the ‘simplification’ mean, for example, picking up the shortest expression among equivalent expressions, or describing an expression using cos and sin terms, or exponential terms? There is no simple answer to this question, and it depends on the context of use. Here is an example with a common equation cos(a)cos(b) – sin(a)sin(b) = cos(a+b). If b is (n-1)*a, almost everyone will

think that cos(n*a) is more simplified than cos(a)cos((n-1)*b) – sin(a)sin((n-1)*a). However, if a = $\omega_a * t_a + \phi_a$, and b = $\omega_b * t_b + \phi_b$ (where ω_x, t_x, and ϕ_x are frequency, time, and phase, respectively), $\cos(\omega_a * t + \phi_a)\cos(\omega_b * t_b + \phi_b) - \sin(\omega_a * t_a + \phi_a)\sin(\omega_b * t_b + \phi_b)$ might be better than $\cos(\omega_a * t_a + \phi_a + \omega_b * t_b + \phi_b)$ in terms of the context of physics.

A simplification process used in a programming language considers many different algorithms to simplify an expression, such as length of the expression, using cos and sin, or exponential functions, etc. If you are doing the simplification manually, you can select the best algorithm to match your purpose. On the other hand, in the case of automatic calculations like the one in this program, it is hard to select the best algorithm at each step of the simplification.

There is a function for the simplification called ‘simplify()’ in SymPy. This function "attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression”, according to the SymPy document. In fact, this function was used in the previous version of this program (ver. 1.1) to handle various cases with a general manner. However, this function sometimes simplifies expressions in unexpected ways. For example, it can’t simplify ‘cos(q)cos(4*q) – sin(q)sin(4q)’ to ‘cos(5q)’. This will be a problem when a rotation of transverse magnetization Ix with an angle q is repeatedly calculated, i.e, 0 + q = q, q + q = 2*q, 2*q + q = 3*q, …, (n-1)*q + q = n*q. Until n = 4, the coefficient of Ix is simplified as cos(n*q). However, when n = 5, cos(q)cos(4*q) – sin(q)sin(4q) is not simplified as cos(5*q). Then, the case with n = 6, the result becomes -24*sin(q)**6 + 36*sin(q)**4 - 27*sin(q)**2/2 - sin(3*q)**2/2 + 1. This type of unexpected behaviors generates long and complicated coefficients through the calculation in this program. Also, as ‘simplify()’ considers many possible algorithms through the simplification, the time cost becomes high.

```
Initial Density Operator: Ix
CS: I q
    Ix*(cos(q)) + Iy*(sin(q))
CS: I q
    Ix*(cos(2*q)) + Iy*(sin(2*q))
CS: I q
    Ix*(cos(3*q)) + Iy*(sin(3*q))
CS: I q
    Ix*(cos(4*q)) + Iy*(sin(4*q))
CS: I q
    Ix*(-sin(q)*sin(4*q) + cos(q)*cos(4*q)) + Iy*((16*sin(q)**4 - 20*sin(q)**2 + 5)*sin(q))
CS: I q
    Ix*(-24*sin(q)**6 + 36*sin(q)**4 - 27*sin(q)**2/2 - sin(3*q)**2/2 + 1) + Iy*(sin(6*q))
```

There are many functions designed for specific cases of simplifications. For example, the function ‘TR8()’ is superior for “Converting products of cos and/or sin to a sum or difference of cos and or sin terms”. It can handle the simplification of cos(a)cos(b) – sin(a)sin(b) ➔ cos(a+b) with any combination of a and b. Also, the time cost is low because it considers a limited way of the simplification. It must be noted that the design of this function is not preferred some NMR calculations.

For example, an expression with ‘simplify()’

```
-sin(o1*t)*cos(pi*J12*t1)*cos(pi*J13*t1)/2
```

becomes

```
-sin(-pi*J12*t1 + pi*J13*t1 + o1*t)/8 - sin(pi*J12*t1 - pi*J13*t1 + o1*t)/8 + sin(pi*J12*t1 + pi*J13*t1
- o1*t)/8 - sin(pi*J12*t1 + pi*J13*t1 + o1*t)/8
```

with 'TR8()'. This result is less readable and lose the context of physics.

As shown above, there is no perfect simplification algorithm (or function) to output our preferred expression always. To consider different purposes of the calculations, three algorithms are introduced starting the version 1.3, namely, 'simplify', 'TR8', and 'fu'. 'simplify' and 'fu' will be fine for general use. The method 'fu' may return better results for the trigonometric simplification. 'TR8' is a subfunction of 'fu'. This can handle the 'cos(a)cos(b) – sin(a)sin(b) ➔ cos(a+b)' type simplification nicely. Also, it is much faster than the other two methods. The drawback of 'TR8' is that the product forms of cos and/sin are converted to the sum or difference of cos and/or sin making coefficients unnecessarily long in some cases.

One of the functions can be set from a **PO class variable** called 'simp'. The default value is simp = 'simplify'. To change the method to 'TR8' or 'fu', use

```
PO.simp = 'TR8'
```

or

```
PO.simp = 'fu'
```

in the command line or codes.

References

Levitt, M. H., *Spin Dynamics*, 2nd Edition, Wiley, 2008.

Keeler, J., *Understanding NMR Spectroscopy*, 1st Edition, Wiley, 2005.

Sørensen, O. W.; Eich, G. W., Levitt, M. H.; Bodenhausen, G.; Ernst, R. R., *Product Operator Formalism for the Description of NMR Pulse Experiments*, *Prog. NMR Spectros.* **1983**, *16*, 163-192.

Güntert, P.; Schaefer, N.; Otting, G.; Wüthrich, K., *POMA: A Complete Mathmatica Implementation of the NMR Product-Operator Formalism*, *J. Magn. Reson. Ser. A*, **1993**, *101*, 103 – 105.

Güntert, P., *Symbolic NMR Product Operator Calculations*, *Int. J. Quant. Chem.* **2006**, *106*, 344 – 350.

SymPy Simplification, https://docs.sympy.org/latest/tutorials/intro-tutorial/simplification.html

SymPy TR8, https://docs.sympy.org/latest/modules/simplify/fu.html#sympy.simplify.fu.TR8

SymPy fu, https://docs.sympy.org/latest/modules/simplify/fu.html#hongguang-fu-s-trigonometric-simplification